by
Jeff Prosise

Tutor

SEGMENTED MEMORY

I've often heard that Intel processors have segmented memory architectures, and that segments limit the amount of code and data a program can have. Can you explain what segmentation is and elaborate upon the pros and cons of a segmented architecture?

Donald E. Whitelock
Turnersville, New Jersey

■ SEGMENTED MEMORY: Although memory in PCs is laid out in one big linear mass, it's accessed as if it were divided into small, discrete blocks. Here's the story of how and why.

In it simplest form, a segment in the Intel architecture is a 64K block of memory. On the 8088, 8086, and 80286 processors, memory in quantities of more than 64K can't be addressed as one flat, linear address space; instead, memory must be broken down into smaller 64K segments and then addressed one ment at a time.

Why *segments?* Intel wanted to enable the 8086 to address a full megabyte of memory. Since the number of different memory addresses you can have is 2^n, where n is the number of bits you're given to represent memory, it was necessary to build a machine that was capable of 20-bit addressing (2^{20} = 1MB).

Intel also wanted to keep its internal registers 16 bits wide for compatibility with earlier Intel microprocessors. This immediately created a serious conflict: 16-bit addressing limits you to 2^{16}, or 64K of RAM. So that both goals could be achieved, Intel opted to give the 8086 20 address lines (permitting up to 2^{20} bytes of data to be addressed) but leave the registers 16 bits wide, and set four of them aside for use as *segment registers*—registers whose sole purpose is to hold segment addresses. Using this scheme, the CPU could form a 20-bit address by combining the contents of a segment register, which holds a paragraph address, with an offset register, which holds a byte address. (*A paragraph* is 16 bytes of memory; thus, you multiply by 16 to translate a paragraph address to a byte address). In other words, the segment register contains the address where the segment begins, and the address in the offset register is interpreted as an offset relative to the base of the segment.

Figure 1 shows how two 16-bit values are combined to form the address of a unique location in memory. The segment value is multiplied by 16 to produce the number 12340h (in hexadecimal notation, all it takes to multiply a number by 16 is to add a zero at the end, just like multiplying by 10 in our decimal numbering system). The offset value 0005h is then added to the result, yielding the absolute address 12345h.

By convention, memory addresses are rarely referred to in absolute form. Instead, they're expressed in segment:offset form, with the segment and offset values written in hexadecimal and separated by a colon. The address 12345h, for example, would normally be written as 1234:0005. The advantage to using this notation is that it clearly spells out the values used to arrive at the resultant address.

That's not the only way to arrive at 12345h, of course. The expression 1234:0005 treats it as the fifth byte in the segment based at paragraph 1234h. But it's also the 21st (hex 15) byte in the segment starting at 1233h (1233:0015), the 37th (hex 25) byte in the segment at 1232h (1232:0025), and so on. For any given byte in memory, there are thousands of different ways to formulate its address. In fact, it's a mistake to believe that memory in a PC equipped with 640K of RAM is divided into ten evenly spaced segments. Segments can begin on any 16-byte boundary. They can also overlap, leaving you 4,096 ways to formulate an address (unless of course it's protected mode, in which case there are as many

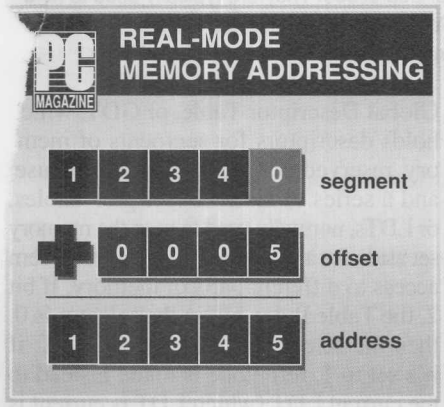


Figure 1: An Intel microprocessor running in real mode forms a 20-bit address by shifting the contents of a 16-bit segment register 4 bits left (to form a paragraph address) and ding a 16-bit offset. Here, the segment dress, 1234h, is paired with the offset address 0005h to form the absolute address 12345h.

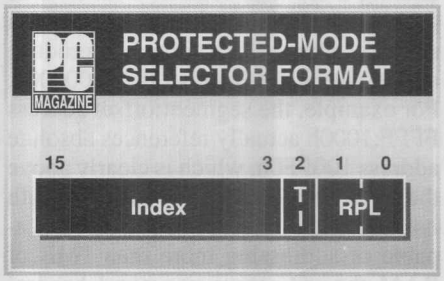


Figure 2: In protected mode, segment registers are loaded with selectors rather than physical segment addresses. The 13-bit index field holds a descriptor number, which references a descriptor in the Global Descriptor Table (GDT) if bit 2 (the Table Index bit) is 0 or in the Local Descriptor Table (LDT) if Table Index is 1. The RPL field holds a number from 0 to 3, representing the requested privilege level, with 0 ranking highest and 3 lowest.

ways to formulate an address as there are paragraphs of memory).

One way to view segments is to think of them as 64K windows onto memory that can be moved around as needed to reference specific targets inside a larger, more extensive address space. On the 8086, registers CS, DS, ES, and SS hold segment addresses. By changing a segment register's value, you can move the 64K window associated with that register to a different location in memory. In assembly language, the instruction

```
MOV AL, ES:[1000]
```

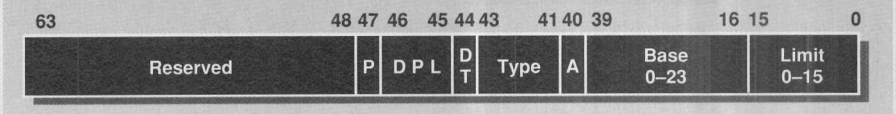
means "go to the segment that ES points to and get me the byte at offset 1000h, then place it in register AL." As you may have guessed, the 64K limit on segment size arises from the fact that the largest number that can be formed with a 16-bit quantity, and thus the highest offset that can be addressed within a segment, is 65,535—exactly 64K minus 1.

286 SEGMENTATION

The same concept of segmentation that applies to the 8086 processor also applies to a 286, 386, or 486 running in real mode. Memory addressing is limited to 1MB, and 20-bit addresses are formed from 16-bit values.

In certain cases, you can access more than 1MB of memory in real mode by loading a segment register with a value greater than F000h and combining it with an offset address high enough to produce an absolute address higher than 100000h. For example, the segment:offset address FFFF:1000h actually references absolute address 100FF0h, which is clearly above the 1MB boundary. Since an 8086, with its 20 address lines, isn't physically capable of addressing more than 1MB of RAM, the address wraps around, becoming 0000:0FF0h. But on 286s, 386s, and 486s, selectively enabling the A20 address line (which is normally used only in protected mode) allows the CPU to reach into the first 64K (minus 16 bytes) of upper memory, even though it's running in real mode. This 64K region is known as the *High Memory Area,* or HMA. Microsoft's prototype Extended Memory Specification (XMS) driver HIMEM.SYS, which was discussed in the December 11, 1990, Tutor column, provides a mechanism for controlling the A20 address line and for arbitrating accesses to the HMA among competing processes.

286 SEGMENT DESCRIPTOR FORMAT



Bit(s)	Description
0–15	16-bit limit (segment size)
16–39	24-bit base address of segment
40	Accessed (A) bit 0 = selector for the descriptor has been transferred into a segment register 1 = selector for the descriptor has not been transferred into a segment register
41–43	Types of operations permitted on the segment 000 = data segment; read-only; expand up 001 = data segment; read/write; expand up 010 = data segment; read-only; expand down 011 = data segment; read/write; expand down 100 = code segment; execute-only; nonconforming 101 = code segment; execute/read; nonconforming 110 = code segment; execute-only; conforming 111 = code segment; execute/read; conforming
44	Descriptor Type (DT) bit 0 = system segment or system gate 1 = memory segment
45–46	Descriptor Privilege Level (DPL) bits 00 = privilege level 0 01 = privilege level 1 10 = privilege level 2 11 = privilege level 3
47	Present (P) bit 0 = segment not present in memory 1 = segment present in memory
48–63	Reserved; should be zeroed for compatibility with 386

Figure 3: The 286 segment descriptor defines the location in memory and the size of a segment in protected mode. Base addresses are 24 bits long, allowing segments to be located anywhere in the 286's 16MB physical address space. Limits are 16 bits long, restricting the maximum segment size to 64K. Bits 40 through 47 store auxiliary information that defines the segment's privilege level, type, and current status. *Expand down* segments are segments that extend downward in memory, from higher to lower addresses. They are typically used to hold stacks, which also grow downward in memory. *Conforming segments* are special code segments that may be called by processes running with equal or lower privilege levels.

But protected mode is quite different. In protected mode, memory is still segmented, but segment registers hold quantities known as *selectors* rather than segment addresses. Selectors reference descriptors stored in descriptor tables located in RAM, and descriptors, in turn, spell out the physical locations (and sizes) of corresponding segments of memory.

Figure 2 shows the format of a selector. The 13-bit index field serves as an index into one of the system's descriptor tables. One table can hold up to 8,192 different descriptors. The operating system typically sets up several such tables—a Global Descriptor Table, or GDT, which holds descriptors for segments of memory reserved for operating system use and a series of Local Descriptor Tables, or LDTs, normally used to map the memory set aside for application programs to govern access to different parts of memory. If bit 2, the Table Index bit, in the selector is 0, then the selector references the GDT; if it's set to 1, reference is made instead to the current LDT (which LDT is current is determined by another register, the LDTR register, located in the CPU).

Bits 0 and 1 hold the Requested Privilege Level, or RPL. Privilege levels play a role in 286/386 memory protection by restricting segment accesses to processes (that is, application programs) with equal or higher privilege levels than the seg-

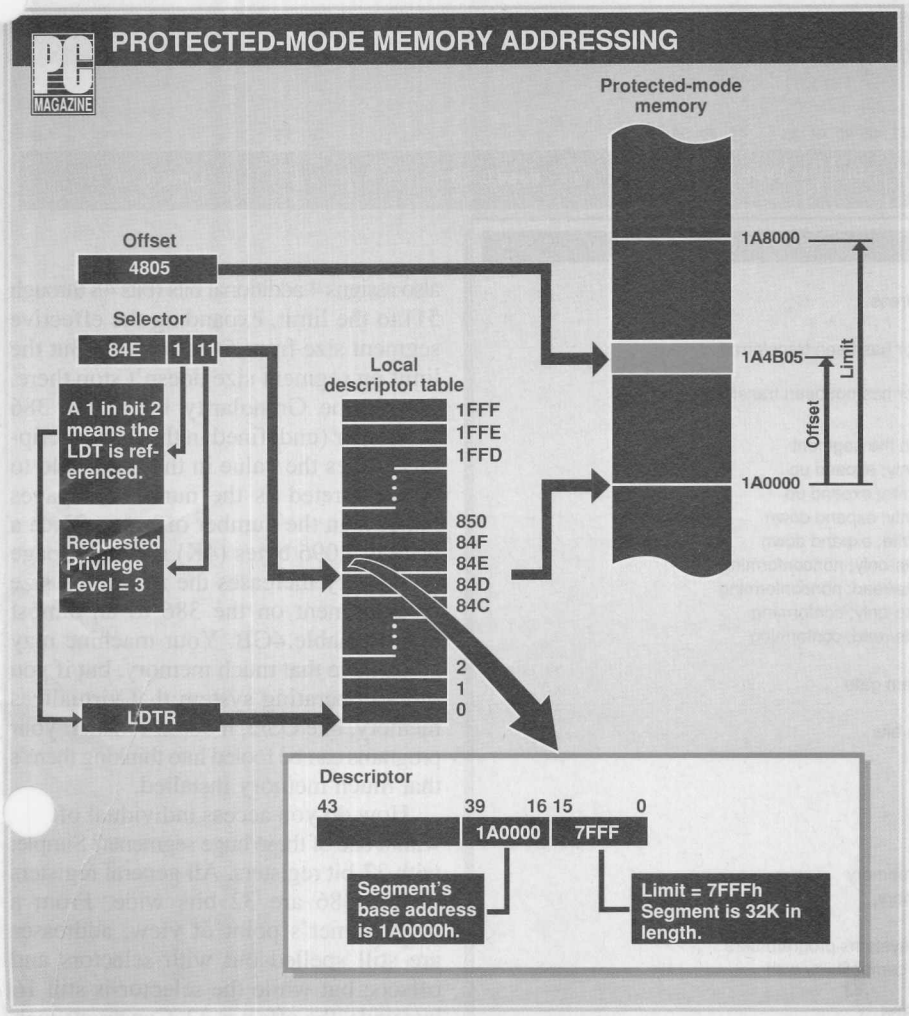


Figure 4: In protected mode, the path to a location in memory is keyed to a descriptor table, where the segment's base address and limit are defined. Here, the selector references descriptor number 84Eh in the LDT. The base address of the segment, 1A0000h, is stored in the descriptor itself, as is the segment's limit, 7FFFh. The offset value 4B05h is added to the base to form absolute address 1A4B05h. The maximum size of the base, limit, and offset is processor-dependent.

ments they're trying to access. In OS/2, for example, regions of memory that belong to the operating system are assigned the highest-possible privilege level, preventing application programs, which run at lower privilege levels, from corrupting operating system data. That's one reason it's called *protected* mode.

Figure 3 shows the format of the 286 segment descriptor. Bit fields within the 64-bit descriptor define where a segment lies in memory and how far it extends. Segment sizes are adjustable on the 286, but the maximum size of a segment is still 64K—the same limit imposed on the 8086. Why? The descriptor's limit field, which defines the size of the segment, is only 16 bits wide. The limit is the highest offset that can be addressed within the segment; any attempt to access memory beyond the posted limit results in a protection fault. On the other hand, 24 bits are set aside for the segment's base address, permitting the segment to be located anywhere within the 286's 16MB physical address space.

Bits 40 through 47 in the 286 segment descriptor contain auxiliary information such as whether or not the selector for the descriptor has been loaded into a segment register, whether the corresponding segment is a code or data segment and what the descriptor type (whether it points to a memory segment or to some other type of system object, such as a call gate, which allows processes to call routines with higher privilege levels), the descriptor privilege level, and whether the segment is currently present in memory are also found here. This type of information, esoteric as it may seem, is crucial to a CPU operating in a multitasking environment where each process must be protected from others active in the system at the same time.

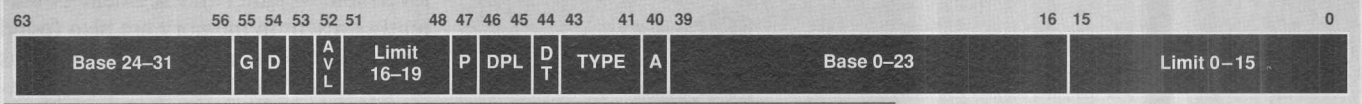
Figure 4 diagrams the process a 286 goes through to convert an address stored in selector:offset form to a physical address in memory. In this example, the selector references a descriptor in the Local Descriptor Table, whose base address is stored in the LDTR register. The selector's index field holds the hexadecimal value 84E, which points to the 84Eth descriptor in the LDT. The descriptor's base address field holds the 24-bit value 1A0000h, which says that the corresponding segment is based at absolute address 1A0000h, a location approximately midway between the 1MB and 2MB marks. The limit field holds 7FFFh, indicating that the segment is 32K in length. Finally, the offset value 4B05h is applied relative to the base of the segment (just as it is in real mode), yielding the absolute address 1A4B05h.

At a glance, all this may seem confusing. But a closer look reveals some of the inherent advantages to structuring memory this way. By setting up a unique LDT for each process, a protected-mode operating system can load multiple processes into memory and keep them physically isolated from one another. The premise is that a process can't access any region of memory that's not mapped into its LDT. The operating system can also prevent processes from accessing the GDT by running them at lower privilege levels than the GDT allows. In the end, you have a solid foundation for a multitasking environment, where no program can corrupt the system by overwriting memory that has been allocated to another program.

386 SEGMENTATION

Segmentation on the 386 is remarkably similar to segmentation on the 286, but with a twist: the addressing capabilities of the 386 are far superior to those of the 286, providing more flexibility in the way protected-mode memory can be structured. In fact, the 386 was the first Intel CPU to

THE 386 SEGMENT DESCRIPTOR FORMAT



Bit(s)	Description
0–15	Lower 16 bits of 20-bit limit
16–39	Lower 24 bits of 32-bit base address
40	Accessed (A) bit 0 = selector for the descriptor has been transferred into a segment register 1 = selector for the descriptor has not been transferred into a segment register
41–43	Types of operations permitted on the segment 000 = data segment; read-only; expand up 001 = data segment; read/write; expand up 010 = data segment; read-only; expand down 011 = data segment; read/write; expand down 100 = code segment; execute-only; nonconforming 101 = code segment; execute/read; nonconforming 110 = code segment; execute-only; conforming 111 = code segment; execute/read; conforming
44	Descriptor Type (DT) bit 0 = system segment or system gate 1 = memory segment
45–46	Descriptor Privilege Level (DPL) bits 00 = privilege level 0 01 = privilege level 1 10 = privilege level 2 11 = privilege level 3
47	Present (P) bit 0 = segment not present in memory 1 = segment present in memory
48–51	Upper 4 bits of 20-bit limit
52	Available (AVL) bit; available to systems programmers
53	Reserved; should be zeroed for compatibility with future microprocessors
54	Default (D) bit; always 1 on the 386
55	Granularity (G) bit 0 = byte granularity 1 = page granularity
56–63	Upper 8 bits of 32-bit base address

Figure 5: The 386 segment descriptor, an extension of the 286 descriptor, permits segments to be located anywhere in a 4GB address space and individual segments to be up to 4GB in length. Although the 20-bit limit field would at first seem to restrict segment sizes to 1MB, setting the descriptor's granularity bit (bit 55) causes the value in the limit field to be interpreted as the number of 4K pages rather than the number of bytes, resulting in a 4GB ceiling on segment size.

remove the 64K size limit on segments, making it an attractive platform for operating systems that fare better on processors not bound by segmented memory, such as Unix.

Figure 5 maps the contents of the 386 segment descriptor, which is simply an extension of the 286 descriptor. Note the downward compatibility between the two formats: every field that's defined on the 286 is also defined on the 386 in the same location. As a result, a program written to run in protected mode on the 286 will run fine on a 386, as long as the unused bits in the descriptor are zeroed out, a precaution Intel recommended to 286 systems programmers from the beginning, to ensure compatibility with future CPUs.

The 386 assigns 8 additional bits (bits 56 through 63) to a segment's base address, expanding the range of memory it can be located in from 16MB to 4GB. It also assigns 4 additional bits (bits 48 through 51) to the limit, expanding the effective segment size from 64K to 1MB. But the limit on segment size doesn't stop there. Setting the Granularity bit in the 386 descriptor (undefined in the 286 descriptor) causes the value in the limit field to be interpreted as the number of pages rather than the number of bytes. Since a page is 4,096 bytes (4K) in length, page granularity increases the maximum size of a segment on the 386 to an almost unfathomable 4GB. Your machine may never have that much memory, but if you run an operating system that virtualizes memory, like OS/2, it doesn't matter: your programs can be fooled into thinking there's that much memory installed.

How do you access individual of within one of these huge segments? Simple: with 32-bit registers. All general registers on the 386 are 32 bits wide. From a programmer's point of view, addresses are still spelled out with selectors and offsets; but while the selector is still 16 bits wide, the offset is 32. Constructs such as ES:[EDI] are common in 386 assembly language, where EDI is the 32-bit version of the 16-bit DI register 8086 and 286 programmers are accustomed to.

VIRTUAL MEMORY

Are segments really all that bad? Once upon a time, when the majority of applications were still written in assembly language, they did make life difficult for developers. Segments permeate every fiber of a program's being. It's inherently more difficult to write a program that deals with multiple code or data segments than one that limits its code and data to 64K—the size of a single segment. That's why, for example, good-old BASICA limited code and data to 64K and some language interpreters still do. Fortunately, compilers capable of handling far code and data objects—code and data stored in remote segments—now do most of the work for us.

But segments aren't always the bad

guys. In protected mode, they play a crucial role in helping operating systems to virtualize memory. Virtual memory requires that physical memory be partitioned into easily managed units that can be swapped to and from the hard disk. Segments 64K or less in length fit this role quite well. On both the 286 and 386, the descriptor's Present bit (bit 47) is set to 1 when a segment is present in memory and to 0 when it is swapped out.

When a process attempts to access a swapped segment, the CPU generates an exception that is trapped by the operating system. The operating system in turn gets the opportunity to load the segment back into memory, swapping other segments out, if necessary, to make room.

The problem with huge segments on the 386 is that they can be too large to be swapped to the hard disk. They can also be small enough to fit on the hard disk but too large to be swapped in and out efficiently. That's why the 386 supports a second form of virtual memory management known as *paging*. Paging divides all of memory into a series of evenly spaced 4K pages. If segment sizes prohibit swapping at the segment level, pages can be swapped instead.

The 386-specific version of OS/2 due out later this year will use paging rather than segmentation to virtualize memory, enabling developers to set up huge 386 segments without sacrificing the virtual-memory capabilities of the CPU.

If you'd like to know more about segmentation, paging, protection, and virtual memory management on the 286 and 386, here's a list of recommended reading: *Inside the 80286,* by Ed Strauss (1986, Prentice-Hall); *Programming the 80386,* by John H. Crawford and Patrick P. Gelsinger (1987, SYBEX); and *The 80386 Book,* by Ross P. Nelson (1988, Microsoft Press).

ASK THE TUTOR

The Tutor solves practical problems and explains techniques for using your hardware and software more productively. Questions about DOS and systems in general are answered here. To have your question answered, write to Tutor, *PC Magazine,* One Park Avenue, New York, NY 10016, or upload it to PC MagNet (see page 8 for access instructions). We're unable to answer questions individually. ■